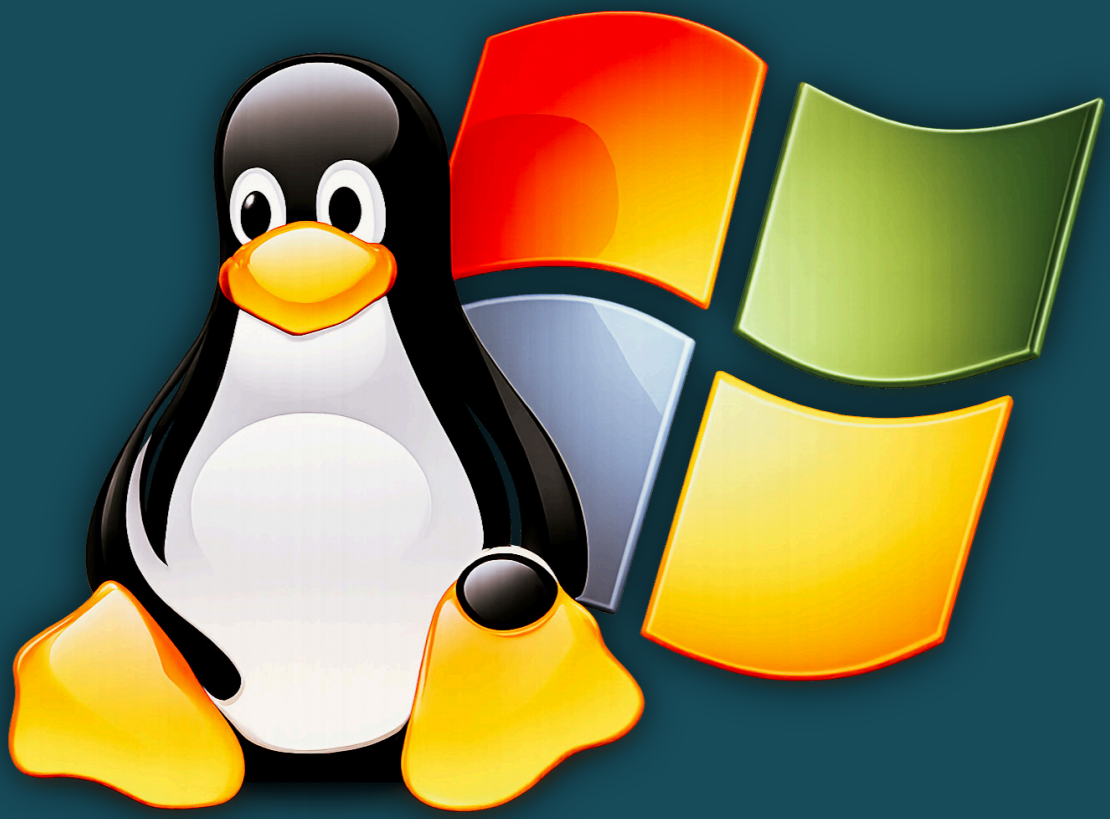
VG



# MCA-103

## OPERATING SYSTEM

# PREFACE

Dear Reader,

It is a privilege to present this comprehensive collection of **RTU MCA Semester Examination Notes**, designed to cover the complete syllabus with clarity and academic accuracy. Every concept, definition, explanation, and example has been organized to support thorough understanding and effective exam preparation.

The purpose of these notes is to simplify complex topics and provide a reliable study resource that can assist in both detailed learning and quick revision. Continuous effort has been made to ensure correctness and relevance; however, learning grows when readers engage, question, and explore further.

May these notes serve as a strong academic foundation and contribute meaningfully to your preparation and future growth.

Warm regards,
**Virendra Goura**
Author
www.virendragoura.com

# DISCLAIMER

This e-book has been created with utmost care, sincere effort, and extensive proofreading. However, despite all attempts to avoid mistakes, there may still be some errors, omissions, or inaccuracies that remain unnoticed.
This e-book is issued with the understanding that neither the author nor the publisher shall be held responsible for any loss, damage, or misunderstanding arising from the use of the information contained within.
All content provided is for educational and informational purposes only.

# MCA-103 Operating System

## Unit-1

**Introduction:** Definition and types of operating systems, Batch Systems, multi programming, timesharing, parallel, distributed and real-time systems, Operating system structure, Operating system components and services, System calls, system programs, system boot.

**Process Management:** Process concept, Process scheduling, Cooperating process, Threads, Interprocess communication, CPU scheduling criteria, Scheduling algorithms, Multiple-processor scheduling and Algorithm evaluation.

## Unit-2

**Process Synchronization and Deadlocks:** The Critical-Section problem, synchronization hardware, Semaphores, Classical problem of synchronization, Critical regions, Monitors, Deadlock-system model, Characterization, Deadlock prevention, Avoidance and Detection, Recovery from deadlock, Combined approach to deadlock handling.

**Storage Management:** Memory Management -Logical and Physical Address Space, Swapping, Contiguous Allocation, Paging, Segmentation with paging, Virtual Memory, Demand paging and its performance, Page replacement algorithms, Allocation of frames, Thrashing, Page Size and other considerations.

## Unit-3

**Introduction to concept of Open Source Software:** Introduction to Linux, Evolution of Linux, Linux vs. UNIX, Different Distributions of Linux, Installing Linux, Linux Architecture, Linux file system (inode, Super block, Mounting and Unmounting), Essential Linux Commands (Internal and External Commands), Kernel, Process Management in Linux, Signal Handling, System call, System call for Files, Processes and Signals.

## Unit-4

**Shell Programming:** Shell Programming - Introduction to Shell, Various Shell of Linux, Shell Commands, I/O Redirection and Piping, Vi and Emacs editor, Shell control statements, Variables, if-then-else, case-switch, While, Until, Find, Shell Meta characters, Shell Scripts, Shell keywords, Tips and Traps, Built in Commands, Handling documents, C language programming, Prototyping, Coding, Compiling, Testing and Debugging, Filters.

## Unit-5

**Linux System Administrations:** File listings, Ownership and Access Permissions, File and Directory types, Managing Files, User and its Home Directory, Booting and Shutting down (Boot Loaders, LILO, GRUB, Bootstrapping, init Process, System services).

# UNIT - 1

**Introduction:** Definition and types of operating systems, Batch Systems, multi programming, timesharing, parallel, distributed and real-time systems, Operating system structure, Operating system components and services, System calls, system programs, system boot.

**Process Management :** Process concept, Process scheduling, Cooperating process, Threads, Interprocess communication, CPU scheduling criteria, Scheduling algorithms, Multiple-processor scheduling and Algorithm evaluation.

## Definition

An **Operating System (OS)** is a **system software** that acts as a **bridge between the user and the computer hardware**.
It manages the execution of application programs and controls the operations of computer hardware.

Formally:

An operating system is a program that manages the computer hardware, provides a basis for application programs, and acts as an intermediary between the computer user and the computer hardware.

The OS provides a **user-friendly environment** to execute programs efficiently.

### Objectives of Operating System

1. **Convenience:** Makes the computer system easy to use.
2. **Efficiency:** Improves the system performance and utilizes hardware efficiently.
3. **Ability to Evolve:** Should be flexible for hardware and software advancements.
4. **Resource Management:** Manages resources like CPU, memory, devices, and data efficiently.
5. **Security & Protection:** Prevents unauthorized access and ensures data integrity.

### Functions of Operating System

| Function | Description | Example |
|---|---|---|
| **Process Management** | Handles process creation, scheduling, synchronization, and termination. | fork(), wait() |
| **Memory Management** | Allocates and deallocates memory spaces to processes. | Virtual memory, paging |
| **File Management** | Manages file operations like creation, deletion, read/ write. | FAT32, NTFS |
| **Device Management** | Manages I/O devices through device drivers. | Keyboard, printer |
| **Storage Management** | Organizes data storage and access on disks. | Disk scheduling |
| **Security and Protection** | Protects data from unauthorized access. | Authentication |

| | | |
|---|---|---|
| **Error Detection** | Detects hardware or software errors. | Parity checks |
| **User Interface** | Provides CLI (Command Line Interface) or GUI. | Windows Explorer, Shell |

**Advantages of Operating System**

1. Efficient resource utilization
2. Provides convenient user interface
3. Ensures system security
4. Enables multitasking
5. Handles errors and recovers from failures

## 2. TYPES OF OPERATING SYSTEMS

### 1. Batch Operating System

Jobs with similar needs are grouped together in batches and executed without user interaction.

**How it works:**

1. Users submit jobs to the operator.
2. Operator groups them into batches.
3. The OS executes jobs one after another.

**Example:** IBM mainframes in the 1960s.

**Advantages:**

- High throughput.
- Reduced idle time.

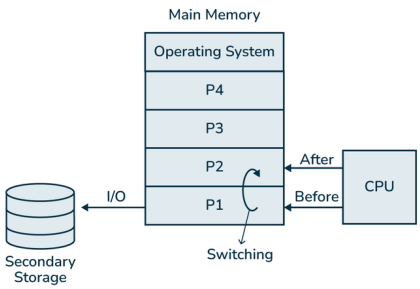
**Disadvantages:**

- No direct user interaction.
- Debugging errors is difficult.
- Long waiting time.

### 2. Multiprogramming Operating System

More than one program resides in memory and CPU switches among them for better utilization.

**How it works:**
When one program waits for I/O, another program uses the CPU.




**Example:** UNIX, Linux.

**Advantages:**

- Better CPU utilization.
- Reduced idle time.
- Efficient performance.

**Disadvantages:**

- Complex memory management.
- Requires large memory.

### 3. Time-Sharing Operating System

Each user gets a small unit of CPU time (time slice). The CPU switches rapidly among users.

**Objective:**
To provide **interactive computing** and **minimize response time**.

**Example:** UNIX Time-Sharing System.

**Advantages:**

- Better response time.
- User interaction supported.

**Disadvantages:**

- High overhead for context switching.
- Requires complex scheduling.

### 4. Parallel Operating System

Multiple processors work together to execute processes simultaneously.

**Used in:** Scientific computations, simulations, supercomputers.

**Advantages:**

- High performance.
- High reliability.

**Disadvantages:**

- Complex design and communication.
- Expensive hardware.

**Example:** Windows HPC, Linux clusters.

**5. Distributed Operating System**

A collection of independent computers appear to the users as a single coherent system.

**Example:** Amoeba, LOCUS, Sprite.

**Advantages:**

- Resource sharing.
- Load balancing.
- Fault tolerance.

**Disadvantages:**

- Complex communication.
- Security issues.

**6. Real-Time Operating System (RTOS)**

Used for applications where the system must respond to inputs instantly.

**Types:**

1. **Hard Real-Time OS:**
   Strict timing constraints (e.g., missile systems, air traffic control).
2. **Soft Real-Time OS:**
   Missed deadlines are tolerable (e.g., multimedia systems).

**Example:** VxWorks, QNX, RTLinux.

**Characteristics:**

- Predictable response.
- Task prioritization.
- Minimal latency.

## 3. OPERATING SYSTEM STRUCTURE

An **Operating System (OS) structure** defines how the various components of an OS (like the kernel, user interface, and hardware drivers) are organized and how they communicate with each other.

Think of it as the "blueprint" for the software. If the OS were a building, the structure would determine where the foundation, plumbing, and rooms go.

Here are the most common OS structures explained:

**A. Monolithic Structure**

In this design, the entire operating system runs as a single, large program in the **kernel space**. It is like a massive Swiss Army knife where every tool is attached to the same handle.

- **How it works:** All services (file management, memory, process scheduling) are packed into one large file.
- **Pros:** Extremely fast because components can talk to each other directly without barriers.
- **Cons:** If one part crashes (e.g., a bad driver), the entire system can crash (Blue Screen of Death).

**B. Layered Structure**

This approach divides the OS into distinct layers. Each layer can only communicate with the layers immediately above and below it.

- **Layer 0:** Hardware (bottom).
- **Layer N:** User Interface (top).
- **Pros:** easy to debug. If an error occurs in Layer 2, you know you don't need to check Layer 4.
- **Cons:** Can be slower because data has to pass through many steps to get from the user to the hardware.

**C. Microkernel Structure**

This design tries to keep the kernel as small ("micro") as possible. It removes non-essential services (like file systems and device drivers) from the kernel and runs them in **user space** (like normal programs).

- **How it works:** The kernel only handles the absolute basics: communication and CPU scheduling. If you want to save a file, the kernel sends a message to a "File Server" program running in user space.
- **Pros:** Very stable. If a driver crashes, the kernel keeps running, and you can just restart the driver.
- **Cons:** Slower performance due to the constant "message passing" between the kernel and user programs.

**D. Hybrid Structure (Modular)**

Most modern operating systems (like **Windows** and **macOS**) use a hybrid approach. They look like a Monolithic kernel (for speed) but allow you to load and unload specific parts (modules) on the fly, like a Microkernel.

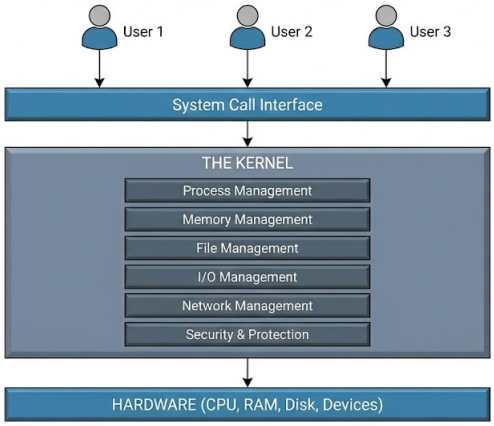
- **How it works:** You have a main kernel, but you can plug in "modules" (like a USB driver) only when needed, without restarting the system.

## 4. OPERATING SYSTEM COMPONENTS

An Operating System is like a manager that handles different departments (components) to keep the computer running smoothly. Each component is responsible for a specific task, like managing the CPU, memory, or files.

Here is a simple text diagram showing how these components sit between the User and the Hardware:

1. **Process Management**




   - **Function:** The OS manages "processes" (running programs). It decides which process gets the CPU and for how long.
   - **Key Tasks:** Creating/deleting processes, pausing/resuming them, and handling synchronization (so two processes don't crash into each other).

2. **Memory Management (RAM)**
   - **Function:** Manages the primary memory (RAM).
   - **Key Tasks:** Keeps track of which parts of memory are currently being used and by whom. It allocates memory when a program starts and frees it when the program ends.

3. **File Management**
   - **Function:** Organizes data into files and directories (folders).
   - **Key Tasks:** Creating, deleting, reading, and writing files. It maps these "files" onto the physical hard drive.

4. **I/O Device Management**
   - **Function:** Manages input/output devices like keyboards, mice, printers, and screens.
   - **Key Tasks:** It uses **Device Drivers** to talk to the hardware. It handles "buffering" (temporarily storing data) so the CPU doesn't have to wait for a slow printer.

5. **Secondary Storage Management**
   - **Function:** Manages the hard disk or SSD (permanent storage).
   - **Key Tasks:** Since Main Memory (RAM) is volatile (data is lost when power is off), the OS manages the disk to store data permanently. It also manages **free space** and **disk scheduling**.

6. **Network Management**
   - **Function:** Manages communication between different computers.
   - **Key Tasks:** Handles protocols (like TCP/IP) to send data across a network or the internet.

7. **Security & Protection**
   - **Function:** Protects the system from unauthorized access.

- **Key Tasks:** Using passwords (Authentication) and ensuring one user's program cannot interfere with another user's data (Protection).

8. **Command Interpreter System (User Interface)**
   - **Function:** The interface between the user and the OS.
   - **Key Tasks:** It reads commands from the user and executes them. This can be a **CLI** (Command Line Interface like cmd) or a **GUI** (Graphical User Interface like Windows).

## 5. OPERATING SYSTEM SERVICES

1. **Program Execution** – Execute and terminate programs.
2. **I/O Operations** – Read/write data from/to devices.
3. **File System Manipulation** – File access and management.
4. **Communication** – Between processes or systems.
5. **Error Detection** – Identify and handle system errors.
6. **Resource Allocation** – Distribute CPU, memory, devices.
7. **Accounting** – Keep track of usage statistics.
8. **Protection and Security** – Prevent unauthorized access.

## 6. SYSTEM CALLS

System calls are **interfaces between user programs and OS kernel functions**.

**Example:**
A user program cannot directly perform I/O; it requests the OS using system calls like read(), write(), etc.

**Types of System Calls:**

| Category | Example |
|---|---|
| Process Control | fork(), exec(), exit() |
| File Management | open(), close(), read(), write() |
| Device Management | ioctl(), read() |
| Information Maintenance | getpid(), alarm() |
| Communication | send(), receive() |

## 7. SYSTEM PROGRAMS

**System Programs** are utility programs provided by the Operating System that help users and programmers interact with the computer system. They do not control hardware directly but use OS services to perform tasks efficiently.

### 1. File Management

- Handle creation, deletion, copying, renaming, and organization of files and directories.
- Manage file permissions and attributes.

**Examples:** File Explorer, ls, cp, rm

### 2. Status Information

- Provide information about system performance and status.
- Display CPU usage, memory usage, date, and time.

**Examples:** Task Manager, top, ps

### 3. File Modification

- Create and modify the contents of files.
- Used mainly for editing text and program files.

**Examples:** Notepad, vi, nano

### 4. Programming Language Support

- Provide tools for program development and execution.
- Include compilers, interpreters, and assemblers.

**Examples:** GCC compiler, Python interpreter, Java compiler

### 5. Program Loading and Execution

- Load programs into memory and start their execution.
- Manage linking and execution of programs.

**Examples:** Loader, Linker, Command Shell

### 6. Communication Programs

- Allow communication between processes and systems.
- Support data transfer and remote access.

**Examples:** Web browsers, Email clients, FTP, SSH

### 7. Background Services (Daemons)

- Run continuously in the background to support system operations.

**Examples:** Print spooler, Network services, System logging

## 8. SYSTEM BOOT

Booting is the process of **starting the computer and loading the operating system into main memory**.

**Steps:**

1. **POST (Power-On Self Test)** – Tests hardware components.
2. **Bootstrap Loader** – Loads OS kernel from disk.
3. **Kernel Initialization** – Initializes system and drivers.
4. **System Processes Start** – Background services start.
5. **User Login Prompt** – Ready for user operations.

# PROCESS MANAGEMENT:

## 1. PROCESS CONCEPT

A **process** is a **program in execution**. It is an active entity compared to a passive program stored on disk.

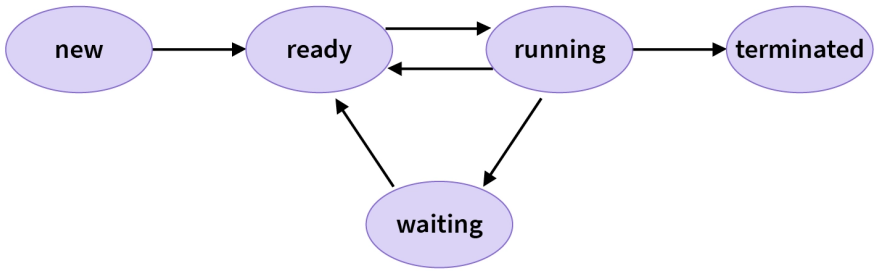
**Process includes:**

- Program code (text section)
- Program counter
- Stack (temporary data)
- Heap (dynamic memory)
- Data section (global variables)

**Process States**

1. **New:** Process being created.
2. **Ready:** Waiting to be assigned to CPU.
3. **Running:** Process currently using CPU.
4. **Waiting/Blocked:** Waiting for an event (I/O completion).
5. **Terminated:** Execution completed.

**State Diagram:**




## 2. PROCESS SCHEDULING

The **scheduler** decides which process runs next.

**Types:**

1. **Long-Term Scheduler:** Chooses which processes are loaded into memory.
2. **Short-Term Scheduler:** Chooses next process for CPU.
3. **Medium-Term Scheduler:** Swaps processes between memory and disk.

**Scheduling Queues**

1. **Job Queue:** All processes in the system.
2. **Ready Queue:** Processes waiting for CPU.
3. **Device Queue:** Processes waiting for I/O device.

## 3. CPU SCHEDULING CRITERIA

**CPU Utilization:**
Percentage of time the CPU remains busy executing processes. Higher CPU utilization indicates efficient use of processor resources and minimal idle time.

**Throughput:**
Number of processes completed by the CPU per unit time. Higher throughput means more work is done in less time.

**Turnaround Time:**
Total time taken by a process from arrival to completion.
*(Turnaround Time = Completion Time − Arrival Time)*

**Waiting Time:**
Total time a process spends waiting in the ready queue before execution.
*(Waiting Time = Turnaround Time − Burst Time)*

**Response Time:**
Time between process submission and the first response from the CPU, especially important for interactive systems.

## 4. CPU SCHEDULING ALGORITHMS

**CPU Scheduling**

CPU Scheduling is an operating system technique used to decide which process from the ready queue should be allocated the CPU next, so that system performance, efficiency, and multitasking capability are improved.

**Need for CPU Scheduling**

CPU scheduling is required because multiple processes compete for the CPU simultaneously. Efficient scheduling ensures fair CPU allocation, reduces waiting and turnaround time, improves response time, and prevents the CPU from remaining idle.

**Types of CPU Scheduling**

CPU scheduling is classified into:

1. **Primitive (Preemptive) Scheduling**
2. **Non-Primitive (Non-Preemptive) Scheduling**

Primitive scheduling allows interruption of a running process, whereas non-primitive scheduling does not allow interruption once execution starts.

## ALGORITHMS

### 1. First Come First Served (FCFS)

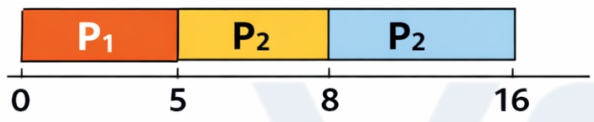
FCFS is the simplest CPU scheduling algorithm in which processes are executed strictly in the order of their arrival in the ready queue. It is easy to implement but may cause the convoy effect.

**Type:**
Non-Primitive (Non-Preemptive)

**Numerical Example:**

| Process | Arrival Time | Burst Time |
|---|---|---|
| P1 | 0 | 5 |
| P2 | 1 | 3 |
| P3 | 2 | 8 |

**Gantt Chart:**




**Waiting Time:**
WT(P1)=0, WT(P2)=4, WT(P3)=6

**Average Waiting Time:**
(0 + 4 + 6) / 3 = **3.33 units**

### 2. Shortest Job First (SJF)

SJF selects the process having the smallest CPU burst time for execution. It provides the minimum average waiting time but requires prior knowledge of CPU burst time.

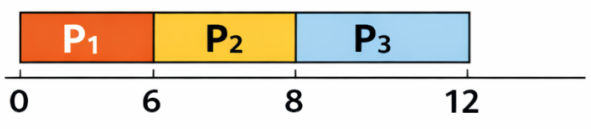
**Types:**

- Non-Primitive SJF
- Primitive SJF (SRTF)

**Numerical Example (Non-Preemptive):**

| Process | Arrival Time | Burst Time |
|---|---|---|
| P1 | 0 | 6 |
| P2 | 1 | 2 |
| P3 | 2 | 4 |

**Gantt Chart**



**Average Waiting Time:**
(0 + 5 + 6) / 3 = **3.67 units**

### 3. Shortest Remaining Time First (SRTF)

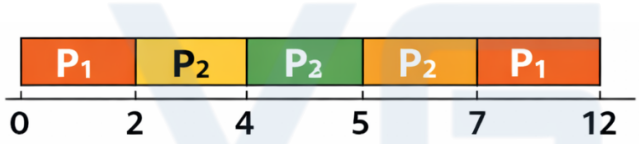
SRTF is the preemptive version of SJF in which the CPU is always allocated to the process with the shortest remaining execution time. It improves response time but increases context switching.

**Type:**
Primitive (Preemptive)

**Numerical Example:**

| Process | Arrival Time | Burst Time |
|---|---|---|
| P1 | 0 | 7 |
| P2 | 2 | 4 |
| P3 | 4 | 1 |

**Gantt Chart:**




### 4. Priority Scheduling

In Priority Scheduling, each process is assigned a priority, and the CPU is allocated to the process with the highest priority. Lower-priority processes may suffer from starvation.

**Types:**

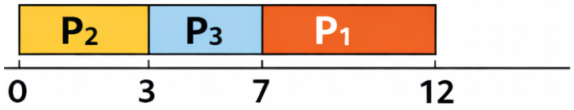
- Primitive Priority Scheduling
- Non-Primitive Priority Scheduling

**Numerical Example (Non-Preemptive):**

| Process | Burst Time | Priority |
|---|---|---|
| P1 | 5 | 3 |
| P2 | 3 | 1 |
| P3 | 4 | 2 |

*(Lower number indicates higher priority)*

**Gantt Chart:**



### 5. Round Robin (RR)

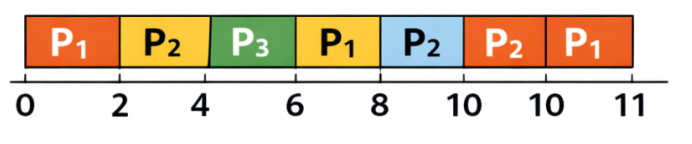
Round Robin scheduling assigns CPU to each process for a fixed time slice called time quantum in a cyclic order. It is widely used in time-sharing systems.

**Type:**
Primitive (Preemptive)

**Numerical Example:**
Time Quantum = 2

| Process | Burst Time |
|---|---|
| P1 | 5 |
| P2 | 4 |
| P3 | 2 |

**Gantt Chart:**




**Average Waiting Time:**
(6 + 6 + 4) / 3 = **5.33 units**

### 6. Multilevel Queue Scheduling

This algorithm divides processes into multiple queues based on priority or process type. Each queue follows its own scheduling algorithm, and movement between queues is not allowed.

### 7. Multilevel Feedback Queue Scheduling

Multilevel Feedback Queue Scheduling allows processes to move between queues based on CPU usage. It prevents starvation and supports dynamic priority adjustment but is complex to implement.

## 5. COOPERATING PROCESSES

**Cooperating Processes** are those processes that **can affect or be affected by other processes executing in the system**.
Such processes require **Inter-Process Communication (IPC)** mechanisms to exchange data and synchronize execution.

Processes can move between queues based on behavior and execution time.
Used in modern OS like Windows and Linux.

- **Independent Processes:** A process is independent if it cannot affect or be affected by the other processes executing in the system. It does not share data with any other process.

- **Cooperating Processes:** A process is cooperating if it can affect or be affected by other processes executing in the system. Clearly, any process that shares data with other processes is a cooperating process.

**Why do we need Cooperating Processes?**

There are four principal reasons for providing an environment that allows process cooperation:

1. **Information Sharing:** Since several users may be interested in the same piece of information (e.g., a shared file), we must provide an environment to allow concurrent access to such information.

2. **Computation Speedup:** If we want a particular task to run faster, we can break it into sub-tasks, each executing in parallel with the others. (Note: This speedup can be achieved only if the computer has multiple processing elements, i.e., CPUs or I/O channels).

3. **Modularity:** We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads.

4. **Convenience:** Even an individual user may work on many tasks at the same time. For instance, a user may be editing, listening to music, and compiling simultaneously.

## 6. INTER-PROCESS COMMUNICATION (IPC)

Cooperating processes require an **IPC mechanism** that will allow them to exchange data and information. There are two fundamental models of IPC:

**A. Shared Memory Model**

In this model, a region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region.

- **Speed:** Faster than message passing because memory access occurs at memory speeds. System calls are required only to establish the shared memory region; once established, no kernel assistance is needed.

- **Conflict:** Major issue is synchronization (ensuring two processes don't write to the same place at the same time).

**B. Message Passing Model**

In this model, communication takes place by means of messages exchanged between the cooperating processes.

- **Mechanism:** Two operations: send(message) and receive(message).

- **Speed:** Generally slower than shared memory because messages are usually implemented using system calls (kernel intervention).

- **Use Case:** useful for exchanging smaller amounts of data and easier to implement for distributed systems (e.g., chat programs).

## 7. THREADS

A thread is the smallest unit of execution in an operating system. It is also known as a lightweight process because it consumes less memory and fewer system resources compared

to a process. A process may contain multiple threads, and each thread performs a specific task within the same process.

**Relationship Between Process and Thread**

A process is a program in execution, whereas a thread is a single sequence of execution within that process. All threads of a process share the same address space, code, data, heap memory, and open files. However, each thread has its own program counter, stack, and register set, which allows independent execution.

**Need for Threads**

Threads are used to improve system performance and responsiveness. In a multithreaded environment, if one thread is waiting for an input/output operation, other threads can continue execution. This results in better CPU utilization and faster program execution.

**Components of a Thread**

Each thread consists of a thread ID, program counter, register set, and stack. These components help the thread to maintain its execution state independently from other threads.

**Types of Threads**

**User-Level Threads**

User-level threads are managed by user-level libraries, and the operating system kernel is not aware of them. These threads are fast to create and manage. However, if one user-level thread is blocked, the entire process may get blocked.

**Kernel-Level Threads**

Kernel-level threads are managed directly by the operating system. The kernel schedules each thread independently, allowing true parallelism on multiprocessor systems. The main drawback is higher overhead due to kernel involvement.

**Advantages of Threads**

Threads provide faster execution, improved responsiveness, efficient resource sharing, reduced overhead, and better utilization of multi-core processors.

**Disadvantages of Threads**

Thread-based programming is complex and difficult to debug. Improper synchronization can cause deadlock, starvation, and priority inversion.

## 8. MULTIPLE PROCESSOR SCHEDULING

Multiple Processor Scheduling is used in systems having **more than one CPU**, where the operating system schedules processes across multiple processors to improve performance, reliability, and parallel execution.

**Need for Multiple Processor Scheduling**

It is required to efficiently distribute processes among multiple CPUs, reduce execution time, increase throughput, and fully utilize all available processors in multiprocessor systems.

**Types of Multiple Processor Scheduling**

**1. Asymmetric Multiprocessing (AMP)**

In asymmetric multiprocessing, **one master processor** handles all scheduling decisions, I/O operations, and system tasks, while other processors execute user processes.

**Example:**
A single CPU assigns tasks to other CPUs.

**Type:**
Centralized scheduling approach.

**2. Symmetric Multiprocessing (SMP)**

In symmetric multiprocessing, **all processors run the operating system scheduler** and share scheduling responsibilities equally.

**Example:**
Modern multicore systems like Linux-based servers.

**Type:**
Decentralized and balanced scheduling approach.

**Advantages of Multiple Processor Scheduling**

It increases system throughput, improves fault tolerance, supports parallel processing, and enhances overall system performance in high-load and real-time environments.

**Disadvantages of Multiple Processor Scheduling**

It increases system complexity, synchronization overhead, and scheduling conflicts, especially when shared resources are accessed simultaneously by multiple processors.

## 9. ALGORITHM EVALUATION

Algorithm Evaluation is the process of **comparing and analyzing CPU scheduling algorithms** to determine their effectiveness based on performance criteria such as waiting time, turnaround time, throughput, and CPU utilization.

**Need for Algorithm Evaluation**

It helps in selecting the most suitable scheduling algorithm for a system based on workload, performance goals, and system requirements.

**Methods of Algorithm Evaluation**

**1. Deterministic Modeling**

Deterministic modeling evaluates scheduling algorithms using **mathematical calculations** with fixed input values.

**Example:**
Calculating average waiting time for given processes.

**Type:**
Analytical evaluation method.

### 2. Queuing Model

Queuing model uses **probabilistic arrival and service rates** to analyze system performance mathematically.

**Example:**
Modeling CPU as a server and processes as customers.

**Type:**
Statistical and mathematical method.

### 3. Simulation

Simulation involves creating a **software model of the system** and running scheduling algorithms on simulated workloads.

**Example:**
Testing FCFS and Round Robin using simulated processes.

**Type:**
Practical and flexible evaluation method.

### 4. Implementation

Implementation evaluates scheduling algorithms by **testing them on a real operating system**.

**Example:**
Implementing a scheduler inside Linux kernel.

**Type:**
Most accurate but costly evaluation method.

# UNIT - 2

**Process Synchronization and Deadlocks**:
The Critical-Section problem, synchronization hardware, Semaphores , Classical problem of synchronization, Critical regions, Monitors, Deadlock-system model, Characterization, Deadlock prevention, Avoidance and Detection, Recovery from deadlock, Combined approach to deadlock handling.
**Storage Management**: Memory Management –Logical and Physical Address Space, Swapping, Contiguous Allocation, Paging, Segmentation with paging, Virtual Memory, Demand paging and its performance, Page replacement algorithms, Allocation of frames, Thrashing, Page Size and other considerations.

## 1. Process Synchronization

When multiple processes execute concurrently and share data, they must be **synchronized** to ensure correct execution.
Process Synchronization is required to maintain **data consistency** and **avoid race conditions**.

### Race Condition

A *race condition* occurs when two or more processes access shared data simultaneously, and the final result depends on the order of execution.

**Example:**
Two processes incrementing a shared counter variable may produce incorrect results if both read and write simultaneously without synchronization.

## 2. The Critical Section Problem

A **critical section** is a part of the program where shared resources are accessed.
The **Critical Section Problem** is to design a protocol so that when one process is executing in its critical section, no other process is allowed to execute in its critical section.

### Structure of a Process

```
do {
   Entry Section       → Request to enter critical section
   Critical Section    → Access shared resources
   Exit Section        → Release lock
   Remainder Section   → Code outside the critical section
} while (true);
```

### Requirements for a Good Solution

1. **Mutual Exclusion:** Only one process can enter the critical section at a time.
2. **Progress:** If no process is in critical section, one waiting process must be allowed to enter.
3. **Bounded Waiting:** Each process gets a fair chance to enter within finite time.

## 3. Synchronization Hardware

Hardware provides **atomic operations** for synchronization.

### Test-and-Set Instruction

A hardware instruction that checks and modifies a lock variable atomically.

### Algorithm using Test-and-Set:

```
do {
   while (TestAndSet(lock));
   // Critical Section
   lock = false;
   // Remainder Section
} while (true);
```

### Disadvantages:

- Busy waiting (CPU cycles wasted).
- May cause starvation.

### Compare-and-Swap Instruction

Another atomic instruction that compares memory contents and swaps if condition matches.
Used in advanced multiprocessor systems for synchronization.

## 4. Semaphores

A **Semaphore** is a synchronization tool introduced by *Edsger Dijkstra*.
It is an integer variable used to control access to shared resources.

### Types of Semaphores

1. **Counting Semaphore:** Integer value ≥ 0. Used to manage a pool of resources.
2. **Binary Semaphore (Mutex):** Value = 0 or 1. Used for mutual exclusion.

### Semaphore Operations

Two atomic operations:

**wait(S)** or **P(S):**

```
wait(S):

   while S <= 0:
      wait;
   S = S - 1;
```

**signal(S)** or **V(S):**

signal(S):

```
S = S + 1;
```

**Example (Mutual Exclusion):**

```
Semaphore mutex = 1;

Process Pi:
do {
  wait(mutex);
  // Critical Section
  signal(mutex);
  // Remainder Section
} while(true);
```

## 5. Classical Problems of Synchronization

These are famous problems used to understand process synchronization concepts.

### a) Bounded Buffer (Producer-Consumer Problem)

- Producer generates items and puts them into a buffer.
- Consumer removes items from the buffer.
- Synchronization needed to avoid buffer overflow and underflow.

**Solution using Semaphores:**

```
Semaphore mutex = 1;
Semaphore empty = n;
Semaphore full = 0;

Producer:
do {
  produce_item();
  wait(empty);
  wait(mutex);
  insert_item();
  signal(mutex);
  signal(full);
} while(true);
```

```
Consumer:
do {
  wait(full);
  wait(mutex);
  remove_item();
  signal(mutex);
  signal(empty);
  consume_item();
} while(true);
```

### b) Readers–Writers Problem

- Multiple readers can read simultaneously.
- Writers require exclusive access.

**Goal:** No data inconsistency when readers and writers operate together.

### c) Dining Philosophers Problem

- Five philosophers sit at a round table and share forks.
- Each philosopher must pick up two forks to eat.
- Improper synchronization can lead to deadlock.

**Solution:** Use semaphores for forks; allow a philosopher to pick up forks only if both are available.

## 6. Critical Regions

A **critical region** is a segment of code in which processes access shared variables.
The compiler ensures that only one process executes within a region at any time.

Used in high-level languages to simplify synchronization.

## 7. Monitors

A **monitor** is a high-level synchronization construct that contains shared variables, procedures, and synchronization between processes.

**Example:**

```
monitor ProducerConsumer {
  condition full, empty;
  procedure produce() {
    if (buffer_full) wait(full);
    insert_item();
    signal(empty);
  }
  procedure consume() {
    if (buffer_empty) wait(empty);
    remove_item();
    signal(full);
  }
}
```

Monitors eliminate low-level semaphore handling and provide safer synchronization.

# ⚠ DEADLOCKS

A *deadlock* is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource held by another process.

## 1. DEADLOCK – SYSTEM MODEL

### Concept of System Model

In an operating system, multiple processes execute concurrently and compete for a limited number of system resources. The **system model** explains how processes request, use, and release resources. Deadlock arises when processes are unable to proceed because of improper resource allocation.

A system consists of:

- **Processes** – active entities that need resources to execute.
- **Resources** – passive entities that are allocated to processes.

### Types of Resources

#### 1. Reusable Resources

Reusable resources are those that can be used by multiple processes one after another and are not destroyed after use.
**Examples:** CPU, memory, printer, disk, files.

#### 2. Consumable Resources

Consumable resources are created and destroyed during execution.
**Examples:** Messages, signals, interrupts.

Deadlock usually occurs with **reusable resources**.

### Resource Usage Cycle

Every process follows the same sequence:

1. **Request** a resource
2. **Use** the resource
3. **Release** the resource

Deadlock occurs when processes **request resources but cannot release the resources they already hold**.

### Resource Allocation Graph (RAG)

A **Resource Allocation Graph** visually represents the allocation and request of resources.

- Process → represented by a circle
- Resource → represented by a rectangle

- Request edge → Process → Resource
- Assignment edge → Resource → Process

If the graph contains a **cycle**, deadlock **may or may not exist** (definite in single-instance resources).

## 2. CHARACTERIZATION OF DEADLOCK

Deadlock can be characterized by four necessary conditions. All four must hold simultaneously for deadlock to occur.

### 1. Mutual Exclusion

At least one resource must be non-shareable, meaning only one process can use it at a time.

**Example:**
Only one process can use a printer at a time.

### 2. Hold and Wait

A process holds one or more resources while waiting to acquire additional resources held by other processes.

**Example:**
Process P1 holds memory and waits for a printer.

### 3. No Preemption

Resources cannot be forcibly taken from a process; they must be released voluntarily.

**Example:**
A file lock cannot be taken away until the process finishes using it.

### 4. Circular Wait

A circular chain of processes exists where each process is waiting for a resource held by the next process.

**Example:**
P1 waits for P2
P2 waits for P3
P3 waits for P1

## 3. DEADLOCK PREVENTION

Deadlock prevention ensures that the system **never enters a deadlock state** by ensuring that **at least one of the necessary conditions is violated**.

**Techniques of Deadlock Prevention**

**1. Elimination of Mutual Exclusion**

- Make resources sharable wherever possible.
- **Limitation:** Not all resources can be shared (printer, disk).

**2. Elimination of Hold and Wait**

- Processes must request **all required resources at once** before execution.
- If resources are not available, the process waits without holding any resource.

**Advantage:** No circular waiting
**Disadvantage:** Poor resource utilization and starvation

**3. Elimination of No Preemption**

- If a process holding resources requests another resource that is unavailable, all its held resources are released.
- The process restarts later.

**Limitation:** Not suitable for non-preemptable resources.

**4. Elimination of Circular Wait**

- Assign a unique ordering to resources.
- Processes must request resources only in increasing order.

**Example:**
If R1 < R2 < R3, then a process cannot request R1 after holding R2.

**Evaluation of Prevention**

- Deadlock never occurs
- System efficiency decreases
- Not preferred for large systems

## 4. DEADLOCK AVOIDANCE

Deadlock avoidance dynamically examines resource allocation requests and grants them only if the system remains in a **safe state**.

**Safe and Unsafe States**

- **Safe State:** There exists at least one safe sequence of processes.
- **Unsafe State:** No safe sequence exists; deadlock may occur.

Safe ≠ Deadlock
Unsafe ≠ Deadlock (but risky)

**Banker's Algorithm (Deadlock Avoidance)**

To determine whether granting a resource request keeps the system in a safe state.

**Data Structures**

- **Available** – Available instances of each resource
- **Max** – Maximum demand of each process
- **Allocation** – Currently allocated resources
- **Need = Max – Allocation**

**Steps of Banker's Algorithm**

1. Check if Request ≤ Need
2. Check if Request ≤ Available
3. Temporarily allocate resources
4. Check for safe sequence
5. If safe → grant request; else → deny

**Example**

If after allocation all processes can finish in some order, the system is safe.

**Advantages**

- Deadlock never occurs
- Better resource utilization than prevention

**Disadvantages**

- Requires advance knowledge of resource needs
- High computational overhead

## 5. DEADLOCK DETECTION

Deadlock detection allows deadlock to occur and then detects it using suitable algorithms.

**Detection Techniques**

**1. Single Instance of Each Resource**

- Use **Wait-For Graph**
- Cycle in the graph indicates deadlock

**2. Multiple Instances of Resources**

- Uses matrices similar to Banker's Algorithm
- If some processes cannot complete → deadlock exists

**Advantages**

- High resource utilization
- Flexible allocation

**Disadvantages**

- Deadlock recovery cost is high
- Deadlock may exist for long duration

## 6. RECOVERY FROM DEADLOCK

Once deadlock is detected, the system must recover.

**Recovery Methods**

**1. Process Termination**

**a) Abort all deadlocked processes**

- Fast recovery
- High loss of work

**b) Abort one process at a time**
Victim selection based on:

- Priority
- Execution progress
- Resources held
- Cost of rollback

**2. Resource Preemption**

- Temporarily take resources from one process
- Assign them to others

Issues involved:

- Selecting victim

- Rollback
- Starvation prevention

## 7. COMBINED APPROACH TO DEADLOCK HANDLING

Modern operating systems use a **combination of prevention, avoidance, detection, and recovery** instead of a single technique.

**Working of Combined Approach**

- Critical system resources → Prevention
- Predictable resource usage → Avoidance
- Non-critical resources → Detection & Recovery
- User-level handling → Ignored or managed by applications

**Example**

- File systems → Prevention
- Databases → Avoidance
- Batch systems → Detection

**Advantages**

- Balanced system performance
- Practical and flexible
- Efficient resource utilization

**Disadvantages**

- Complex design
- Requires careful policy decisions

# STORAGE MANAGEMENT

Memory management is the functionality of an operating system which handles or manages primary memory and moves processes back and forth between main memory and disk during execution.

## 1. Logical and Physical Address Space

Understanding the difference between the address viewed by the CPU and the actual address in memory is the foundation of memory management.

- **Logical Address (Virtual Address):** An address generated by the CPU during program execution. It is the address seen by the user program.
- **Physical Address:** The actual address of a location in the main memory (RAM).
- **Memory Management Unit (MMU):** A hardware device that maps logical addresses to physical addresses at run time.

**Key Differences**

| Feature | Logical Address | Physical Address |
|---|---|---|
| **Visibility** | User can see this. | User cannot see this directly. |
| **Generation** | Generated by the CPU. | Computed by MMU. |
| **Space** | Called Logical Address Space. | Called Physical Address Space. |

Example:

Imagine a book. The "Page 5" written in the Table of Contents is the Logical Address. The physical location where that page sits on the library shelf (e.g., Shelf 3, Slot 12) is the Physical Address.

## 2. Swapping

Swapping is a mechanism in which a process can be temporarily moved out of main memory (RAM) to a secondary storage (disk) and make that memory available to other processes.

**How it Works**

1. **Swap Out:** A process is moved from RAM to the Hard Disk (Backing Store).
2. **Swap In:** Later, the process is brought back from the Disk to RAM to continue execution.

**Use Case**

- It allows the OS to run more processes than can fit into the physical memory at one time.
- **Context Switch Time:** Swapping increases context switch time significantly because moving data between disk and RAM is slow.

## 3. Contiguous Allocation

In this method, each process is contained in a single contiguous section of memory.

### Types of Partitioning

1. **Fixed Partitioning:** Memory is divided into fixed-sized partitions. One process per partition.
    - *Drawback:* **Internal Fragmentation** (wasted space inside a partition if the process is smaller than the partition).
2. **Variable Partitioning:** Partitions are created dynamically according to the size of the process.
    - *Drawback:* **External Fragmentation** (total free space is enough for a process, but it is not contiguous).

### Allocation Strategies

- **First Fit:** Allocate the first hole that is big enough. (Fastest)
- **Best Fit:** Allocate the smallest hole that is big enough. (Reduces wasted space but is slower)
- **Worst Fit:** Allocate the largest hole available. (Leaves large holes, but often inefficient)

## 4. Paging

Paging is a memory management scheme that eliminates the need for contiguous allocation of physical memory.

### Concept

- **Frames:** Physical memory is broken into fixed-sized blocks called **Frames**.
- **Pages:** Logical memory (process) is broken into blocks of the same size called **Pages**.

### How it Works

When a process is to be executed, its **pages** are loaded into any available memory **frames**. The OS maintains a **Page Table** to keep track of which page is stored in which frame.

- **Logical Address** = Page Number ($p$) + Page Offset ($d$)
- **Page Table:** Maps Page Number ($p$) to Frame Number ($f$).

Advantage: No external fragmentation.

Disadvantage: Internal fragmentation (in the last page of a process).

## 5. Segmentation with Paging

Segmentation breaks memory into logical pieces (segments) like "Code," "Stack," and "Data" rather than arbitrary fixed sizes. However, pure segmentation suffers from external fragmentation.

### Solution: Segmentation with Paging

- The logical address space is divided into **Segments**.
- Each segment is then divided into **Pages**.

- This combines the user-friendly view of Segmentation with the efficient memory usage of Paging.

## 6. Virtual Memory

Virtual Memory is a technique that allows the execution of processes that are not completely in memory. It creates an illusion for users that they have a very large main memory.

### Key Benefits

- **Large Programs:** You can run a 4GB game on a computer with only 2GB of RAM.
- **High Degree of Multiprogramming:** More programs can be in the mix since only parts of them need to be loaded.

## 7. Demand Paging and Performance

Demand paging is the most common implementation of virtual memory.

A page is not loaded into main memory until it is explicitly required (demanded) by the CPU. This is often called "Lazy Swapper."

### Page Fault

A **Page Fault** occurs when a program tries to access a page that is not currently in main memory.

1. **Trap:** The OS interrupts the process.
2. **Check:** Check if the reference is valid.
3. **Fetch:** Find a free frame and load the page from disk.
4. **Update:** Update the page table.
5. **Restart:** Restart the instruction that caused the fault.

### Performance

Performance is heavily reliant on the Effective Access Time (EAT).

$$EAT = (1 - p) \times \text{Memory Access Time} + p \times \text{Page Fault Service Time}$$

- Where $p$ is the probability of a page fault.
- **Goal:** Keep $p$ extremely low (close to 0).

## 8. Page Replacement Algorithms

When a page fault occurs and there are no free frames in memory, the OS must swap out an existing page to make room. The algorithm decides *which* page to remove.

### 1. FIFO (First-In, First-Out)

- **Logic:** Replace the oldest page in memory.
- **Pros:** Simple to implement.
- **Cons:** Suffers from **Belady's Anomaly** (adding more frames can sometimes cause *more* page faults).

**2. Optimal Page Replacement (OPT)**

- **Logic:** Replace the page that will not be used for the longest period of time.
- **Pros:** Lowest possible page fault rate.
- **Cons:** Impossible to implement practically because we cannot predict the future. Used as a benchmark.

**3. LRU (Least Recently Used)**

- **Logic:** Replace the page that has not been used for the longest time.
- **Pros:** Good performance, avoids Belady's anomaly.
- **Cons:** Requires hardware support (counters or stacks) to track usage history.

## 9. Allocation of Frames

How many frames do we give to each process?

1. **Minimum Number of Frames:** Defined by the computer architecture (e.g., an instruction might need at least 2 frames to execute).
2. **Equal Allocation:** Split $m$ frames among $n$ processes equally ($m/n$).
3. **Proportional Allocation:** Allocate frames based on the size of the process (larger process = more frames).

## 10. Thrashing

Thrashing is a severe performance degradation state.

A process is **thrashing** if it is spending more time paging (swapping pages in and out) than executing.

**Cause**

- If a process does not have enough frames to hold its frequently used pages (working set), it will fault, replace a page, and immediately fault again because it needed the page it just replaced.
- This causes CPU utilization to drop drastically. The OS sees low CPU usage and tries to add *more* processes, making the problem worse.

**Solution**

- **Working Set Model:** Ensure every process is allocated enough frames to hold its current "locality" of reference.

## 11. Page Size and Other Considerations

**Page Size**

- **Small Pages:**
  - *Pros:* Less internal fragmentation.
  - *Cons:* Larger page tables (more overhead).
- **Large Pages:**
  - *Pros:* Smaller page tables, faster I/O (disk transfers are efficient).
  - *Cons:* More internal fragmentation.

- *Trend:* Modern systems are moving toward larger page sizes (e.g., 4KB to 2MB or even 1GB huge pages) to accommodate larger RAM sizes.

**TLB (Translation Lookaside Buffer)**

- A special, small, fast-lookup hardware cache.
- It stores recent logical-to-physical address translations.
- If a page number is in the TLB (**TLB Hit**), translation is instant. If not (**TLB Miss**), the system must consult the slower page table in RAM.

**Inverted Page Table**

- Instead of one table per process, there is one global table with one entry per *physical frame*.
- Reduces memory needed for page tables but makes searching for a specific logical page harder (hashing is usually used).

# UNIT - 3

**Introduction to concept of Open Source Software:** Introduction to Linux, Evolution of Linux, Linux vs. UNIX, Different Distributions of Linux, Installing Linux, Linux Architecture, Linux file system (inode, Super block, Mounting and Unmounting), Essential Linux Commands (Internal and External Commands), Kernel, Process Management in Linux, Signal Handling, System call, System call for Files, Processes and Signals.

## Introduction to Open Source Software & Linux

## 1. Concept of Open Source Software (OSS)

**Open Source Software (OSS)** refers to software whose source code is made available to the public. The copyright holder grants users the right to study, change, and distribute the software to anyone and for any purpose.

### Core Philosophy: The "Free" in Freedom

In the world of OSS, "Free" doesn't just mean zero cost (gratis); it refers to **liberty** (libre).

- **Freedom 0:** The freedom to run the program as you wish, for any purpose.
- **Freedom 1:** The freedom to study how the program works and change it.
- **Freedom 2:** The freedom to redistribute copies so you can help others.
- **Freedom 3:** The freedom to distribute copies of your modified versions to others.

### Key Differences: Proprietary vs. Open Source

| Feature | Open Source Software (OSS) | Proprietary (Closed Source) |
|---|---|---|
| **Source Code** | Publicly available. | Kept secret (Trade Secret). |
| **Modification** | Allowed and encouraged. | Strictly prohibited; reverse engineering is illegal. |
| **Cost** | Usually free (e.g., Linux, VLC). | Paid license (e.g., Windows, Photoshop). |
| **Support** | Community-driven (Forums, Wikis). | Dedicated customer support team. |
| **Examples** | Linux, Android, Python, Mozilla Firefox. | MS Windows, macOS, Adobe Suite, Oracle DB. |

## 2. Introduction to Linux

### What is Linux?

Strictly speaking, **Linux is just a Kernel**—the core component of the operating system that manages hardware. However, in common usage, "Linux" refers to the complete operating system, which includes:

1. **The Linux Kernel:** (Manages hardware).
2. **GNU Utilities:** (Compilers, shell, file tools).

3. **Desktop Environment:** (GUI like GNOME or KDE).
4. **Application Software.**

**Evolution of Linux (History)**

- **1969 (The Precursor): UNIX** was developed at AT&T Bell Labs by Ken Thompson and Dennis Ritchie. It was powerful but expensive and proprietary.
- **1983 (The Vision): Richard Stallman** launched the **GNU Project**. His goal was to create a completely free UNIX-like operating system. He built the tools (compiler, text editor, shell) but lacked the "Kernel" (the heart).
- **1991 (The Birth): Linus Torvalds**, a student at the University of Helsinki, created a free kernel as a hobby project because he couldn't afford UNIX. He named it **Linux**.
- **The Marriage:** The GNU tools were combined with the Linux Kernel to create the full OS, technically called **GNU/Linux**.

## 3. Linux vs. UNIX

While Linux looks and acts like UNIX, they are different.

| Parameter | Linux | UNIX |
|---|---|---|
| **Origin** | Open Source clone written from scratch by Linus Torvalds. | Developed by AT&T Bell Labs (System V) and Berkeley (BSD). |
| **Licensing** | GPL (General Public License) - Free. | Proprietary licenses - Expensive. |
| **Portability** | Runs on almost any hardware (Watches, Routers, PC, Supercomputers). | Often hardware-specific (e.g., AIX runs on IBM Power servers). |
| **File System** | Supports many (ext4, xfs, btrfs, ntfs). | Supports native UNIX filesystems (zfs, ufs, jfs). |
| **Target Audience** | Everyone (Developers, Home Users, Cloud). | High-end servers, Mainframes, Banks. |

## 4. Different Distributions of Linux (Distros)

Since the Linux Kernel is open source, anyone can take it, package it with different software, and release it. These "flavors" are called **Distributions**.

**Major Families**

1. **Debian Family:**
    - **Debian:** The rock-solid, stable grandfather.
    - **Ubuntu:** Based on Debian. Most popular for beginners and developers. Uses `.deb` packages and `apt` manager.
    - **Kali Linux:** Designed for cybersecurity and penetration testing.
2. **Red Hat Family:**
    - **RHEL (Red Hat Enterprise Linux):** Commercial, paid support, standard for corporate servers.
    - **Fedora:** The testing ground for RHEL. Bleeding-edge features.
    - **CentOS/Rocky Linux:** Free versions of RHEL. Uses `.rpm` packages and `yum`/`dnf` manager.
3. **Android:**
    - Yes, Android runs on a modified Linux Kernel!

## 5. Installing Linux (General Concept)

Installing Linux usually involves these high-level steps:

1. **Download ISO:** Get the image file of the distro (e.g., Ubuntu).
2. **Create Bootable Media:** Burn it to a USB stick.
3. **Partitioning:** Dividing the hard disk. Linux typically needs:
    - **Root Partition (`/`):** Where the OS lives.
    - **Swap Partition:** Virtual memory (used when RAM is full).
4. **Bootloader Setup (GRUB):** Installing the program that loads the OS when you turn on the PC.
5. **User Setup:** Creating a root password and a standard user account.

## 6. Linux Architecture

The system is built in layers, separating the user from the complex hardware.

1. **Hardware Layer:** Physical devices (RAM, HDD, CPU).
2. **Kernel Layer:** The core program that interacts directly with hardware. It hides hardware complexity from applications.
3. **Shell Layer:** An interface (command line) that takes user inputs and sends them to the kernel.
4. **Application/Utility Layer:** User programs (Web Browser, Text Editor, Compilers).

## 7. Linux File System

Linux does not use drive letters (like C: or D:). It uses a **Single Root Hierarchical Tree**.

**The Root (`/`) and Essential Directories**

- `/bin` & `/sbin`: **Binaries**. Essential command programs (ls, cp, ip).

- /etc: **Etcetera**. System configuration files.
- /home: **Home**. Contains personal folders for users (e.g., /home/john).
- /root: The home directory for the Superuser (Administrator).
- /tmp: **Temporary** files (deleted on reboot).
- /dev: **Devices**. Hardware files (e.g., /dev/sda is the hard disk).
- /proc: **Process**. A virtual filesystem containing info about running processes.

**Technical Concepts**

**A. Inode (Index Node)**

The Inode is a data structure on the disk that describes a file.

- **What it stores:** File type, Permissions, Owner ID, Group ID, File Size, Access times, and **pointers to the data blocks** on the disk.
- **What it DOES NOT store:** The **Filename**. (The filename is stored in the directory, pointing to the inode number).
- *Command:* ls -i shows inode numbers.

**B. Super Block**

This is the metadata of the **entire file system**.

- It stores: Total size of the file system, Block size, Total number of inodes, Free block count.
- **Critical:** If the Super Block is corrupted, the OS cannot read the file system at all.

**C. Mounting and Unmounting**

- **Mounting:** Before you can use a storage device (like a USB or Hard Drive partition), it must be attached to a directory in the file tree.
  - *Command:* mount /dev/sdb1 /media/usb
- **Unmounting:** Safely detaching it to ensure data is written.
  - *Command:* umount /media/usb

## 8. Essential Linux Commands

Commands are split into two types based on where they live.

**A. Internal Commands**

- **Definition:** Built directly into the Shell program. They load instantly because they are part of the shell's memory.
- **Examples:**
  - cd (Change Directory)
  - echo (Display text)
  - pwd (Print Working Directory)
  - exit (Close shell)

**B. External Commands**

- **Definition:** These are separate binary files stored on the hard disk (usually in /bin). The shell has to find them and load them.

- **Examples:**
  - ls (List files)
  - cp (Copy files)
  - cat (Read file content)
  - mv (Move/Rename)

## 9. The Kernel: The Core of Linux

The **Kernel** is the central module of an operating system. It is the first program loaded into memory when the computer starts (after the bootloader) and remains in memory until the OS is shut down. It acts as a bridge between the **User Applications** (software) and the **Data Processing** (hardware).

### Kernel Architecture Types

There are two main theories on how to build a kernel. Linux uses the first one.

1. **Monolithic Kernel (Used by Linux):**
   - **Theory:** All OS services (File system, Memory management, Device drivers, Process scheduling) run in the same memory space as the Kernel itself (called "Kernel Space").
   - **Advantage: Speed.** Since all services are in the same place, they can talk to each other directly without complex message passing.
   - **Disadvantage: Crash Risk.** If a bad device driver crashes, it can crash the entire operating system.
2. **Microkernel (Used by Minix/QNX):**
   - **Theory:** The kernel is kept tiny. Most services (like file systems and drivers) run in "User Space" just like normal programs.
   - **Advantage:** Stability. If a driver crashes, the system stays alive.
   - **Disadvantage:** Slower performance due to high communication overhead between modules.

### Core Responsibilities of the Linux Kernel

1. **Process Management:** It decides which process uses the CPU and for how long (Scheduling).
2. **Memory Management:** It keeps track of how much RAM is used and handles "Swapping" (moving inactive data to the hard disk).
3. **Device Management:** It controls peripherals (Mouse, Disk, Keyboard) via **Device Drivers**.
4. **System Calls:** It provides an API (Application Programming Interface) for programs to ask for services.

## 10. Process Management in Linux

A **Process** is more than just a program code. It is an "active" entity.

- **Program:** Passive entity (stored on disk).
- **Process:** Active entity (loaded in RAM with a program counter, stack, and data section).

### The Process Control Block (PCB)

The OS must track every process. It uses a data structure called the Process Control Block (PCB) or task_struct in Linux.

What is inside a PCB?

- **PID (Process ID):** Unique number for identification.
- **Process State:** Is it running, waiting, or sleeping?
- **Program Counter:** Pointer to the next instruction to execute.
- **CPU Registers:** Saved data when the process is paused.
- **Memory Limits:** Which part of RAM belongs to this process.
- **Open Files List:** Which files this process is reading/writing.

### Parent and Child Processes

- **Parent:** The process that creates another process.
- **Child:** The new process created.
- **The `init` Process (or `systemd`):** The very first process started by the Kernel. It has **PID 1**. All other processes in Linux are descendants of `init`.

### Process States (The 5-State Model)

1. **New:** The process is being created.
2. **Ready:** The process is loaded in RAM, waiting in a queue for the CPU scheduler to pick it.
3. **Running:** The process has control of the CPU and is executing instructions.
4. **Blocked / Waiting:** The process cannot run because it is waiting for an event (e.g., waiting for the user to press a key or a file to finish downloading).
5. **Terminated:** The process has finished its job and memory is removed.

## 11. Signal Handling

A **Signal** is a limited form of inter-process communication (IPC) used in Unix-like systems. It is an asynchronous notification sent to a process to notify it of an event. Think of it as a "software interrupt."

### Sources of Signals

1. **Hardware:** (e.g., Division by zero error, Illegal memory access).
2. **The Kernel:** (e.g., Notifying a process that I/O is available).
3. **Users:** (e.g., Pressing `Ctrl+C` or `Ctrl+Z`).
4. **Other Processes:** (e.g., Using the `kill` command).

### How a Process Handles a Signal

When a signal is delivered, the process has three choices:

1. **Ignore it:** The process pretends nothing happened. (Note: `SIGKILL` and `SIGSTOP` cannot be ignored).
2. **Catch it (Trap):** The process executes a specific function (Signal Handler) to handle the event gracefully (e.g., saving a file before closing).

3. **Default Action:** Let the Kernel do the default action (usually terminating the process).

**Important Signal Types**

- **SIGINT (2):** Interrupt from keyboard (Ctrl+C). Action: Terminate.
- **SIGQUIT (3):** Quit from keyboard (Ctrl+\). Action: Terminate and dump core (save memory state for debugging).
- **SIGKILL (9):** Force Kill. Action: Immediate termination. Cannot be blocked or handled.
- **SIGALRM (14):** Timer signal. Sent when a timer set by alarm() expires.

## 12. System Calls

**Concept: User Mode vs. Kernel Mode**

For security, Linux separates memory into two zones:

- **User Space:** Where your normal programs (Web browser, Text editor) run. They have **no access** to hardware.
- **Kernel Space:** Where the core OS runs. It has **full access** to hardware.

If a user program wants to write to the hard disk, it cannot do it directly. It must ask the Kernel to do it. This request is called a **System Call**.

**A. System Calls for File Management**

These calls allow programs to manipulate files.

1. **creat(name, mode)**
   - **Theory:** Creates a new empty file.
   - **Arguments:** name is the filename; mode defines permissions (rwx).
   - **Returns:** A File Descriptor (integer) on success, or -1 on error.
2. **open(name, mode)**
   - **Theory:** Opens an existing file for reading, writing, or both.
   - **Modes:** O_RDONLY (Read only), O_WRONLY (Write only), O_RDWR (Read/Write).
3. **read(fd, buffer, count)**
   - **Theory:** Reads data from the file identified by fd (File Descriptor).
   - **Process:** It copies count bytes from the file into the memory buffer.
   - **Returns:** Number of bytes actually read (0 indicates End of File).
4. **write(fd, buffer, count)**
   - **Theory:** Writes data from the memory buffer to the file fd.
5. **lseek(fd, offset, whence)**
   - **Theory:** Moves the file pointer (cursor) inside a file without reading/writing.
   - **Example:** "Skip the first 100 bytes and start reading."

**B. System Calls for Process Management**

These are the most critical calls for an OS exam.

1. **fork()**
    - **Theory:** This is the **only** way to create a new process in Unix/Linux.
    - **Behavior:** It creates an exact duplicate of the current process.
        - The original process is the **Parent**.
        - The new copy is the **Child**.
    - **Return Values (Crucial for Exams):**
        - It returns **0** to the Child process.
        - It returns the **Child's PID** (a positive integer) to the Parent process.
        - It returns **-1** if the creation failed.
2. **exec() Family (execl, execv, etc.)**
    - **Theory:** fork() only creates a copy. exec() is used to **replace** that copy with a new program.
    - **Example:** When you type ls in the terminal:
        - Terminal calls fork() to create a copy of itself.
        - The Child process calls exec("ls").
        - The Child's memory is wiped and replaced with the code for ls.
3. **wait()**
    - **Theory:** Used by the Parent process to pause execution until one of its Child processes finishes.
    - **Why use it?** To prevent "Zombie Processes." The parent waits to collect the exit status of the child.
4. **exit(status)**
    - **Theory:** Terminates the process normally.
    - **Status:** An integer sent back to the parent (usually 0 means success, non-zero means error).
5. **getpid() and getppid()**
    - getpid(): Returns the process's own ID.
    - getppid(): Returns the **Parent's** process ID.

**C. System Calls for Signals**

1. **kill(pid, sig)**
    - Sends a signal sig to the process with ID pid.
    - **Misconception:** Despite the name "kill," it can send *any* signal (like Pause or Stop), not just death.
2. **alarm(seconds)**
    - Sets a timer. When the timer counts down to zero, the kernel sends the SIGALRM signal to the process.
3. **pause()**
    - Suspends the process (puts it to sleep) until a signal is received. It does not use CPU while waiting.

# UNIT - 4

**Shell Programming:** Shell Programming - Introduction to Shell, Various Shell of Linux, Shell Commands, I/O Redirection and Piping, Vi and Emacs editor, Shell control statements, Variables, if-then-else, case-switch, While, Until, Find, Shell Meta characters, Shell Scripts, Shell keywords, Tips and Traps, Built in Commands, Handling documents, C language programming, Prototyping, Coding, Compiling, Testing and Debugging, Filters.

## 1. Introduction to Shell

The **Shell** is a command-line interpreter that acts as the interface between the **User** and the **Kernel**. It is a program that takes commands from the keyboard, interprets them, and passes them to the operating system to perform.

### The Shell Life Cycle (How it works)

When you type a command (e.g., `ls -l`), the shell goes through these steps:

1. **Read:** It reads the command from Standard Input (Keyboard).
2. **Parse:** It breaks the command into the program name (`ls`) and arguments (`-l`).
3. **Search:** It looks for the program in the directories listed in the `$PATH` variable.
4. **Execute:** It asks the Kernel to run the program.
5. **Output:** It displays the result on the screen.
6. **Loop:** It returns the prompt (e.g., `$`) and waits for the next command.

## 2. Various Shells of Linux

Linux supports multiple shells. Users can switch between them, but `bash` is the standard.

1. **sh (Bourne Shell):** The original Unix shell. It is small and fast but lacks modern features like history or tab completion. Usually found as `/bin/sh`.
2. **bash (Bourne Again Shell):** The default shell for most Linux distributions. It is backward compatible with `sh` but adds features like **command history**, **command line editing**, and **aliases**.
3. **csh (C Shell):** Uses syntax similar to the C programming language. Popular among C programmers but less robust for scripting.
4. **ksh (Korn Shell):** Combines the best features of `sh` and `csh`. It was the standard for commercial Unix (like AIX and Solaris).
5. **zsh (Z Shell):** A modern, feature-rich shell with advanced auto-completion and themes.

## 3. I/O Redirection and Piping

Linux treats all input and output as streams of data. By default, these streams are connected to your terminal, but you can redirect them.

### Standard Streams (File Descriptors)

Every process gets three open channels:

- **0: Standard Input (stdin):** Default source of input (Keyboard).

- **1: Standard Output (stdout):** Default destination for output (Screen).
- **2: Standard Error (stderr):** Default destination for error messages (Screen).

**Redirection Operators**

- `>` **(Output Redirection):** Sends output to a file. Overwrites existing data.
  - *Ex:* `ls > list.txt` (Saves directory list to a file).
- `>>` **(Append Redirection):** Adds output to the end of a file.
  - *Ex:* `date >> log.txt` (Adds timestamp without deleting old logs).
- `<` **(Input Redirection):** Feeds a file into a command.
  - *Ex:* `wc -l < list.txt` (Counts lines in the file).
- `2>` **(Error Redirection):** Redirects *only* error messages.
  - *Ex:* `gcc program.c 2> errors.log` (Saves compilation errors to a file).

**Piping (`|`)**

The Pipe connects the **Standard Output** of the first command to the **Standard Input** of the second command. It allows chaining tools together.

- **Syntax:** `Command_A | Command_B`
- **Example:** `cat file.txt | grep "error" | sort`
  - (Read file → Search for "error" → Sort the results).

## 4. Vi and Emacs Editors

To write scripts or C programs, you need a text editor.

**Vi / Vim (Visual Editor)**

- **Type:** Modal Editor (Different modes for typing and commanding).
- **Ubiquity:** It is installed on essentially every Linux system in the world.
- **Three Modes:**
  1. **Command Mode:** The default. Keys perform actions (e.g., `dd` deletes a line, `yy` copies/yanks a line, `p` pastes).
  2. **Insert Mode:** Press `i` to enter. Used for typing text. Press `Esc` to leave.
  3. **Last Line (Ex) Mode:** Press `:` to enter. Used to save (`w`), quit (`q`), or force quit (`q!`).

**Emacs**

- **Type:** Modeless Editor.
- **Philosophy:** "An operating system inside an editor." Highly extensible.
- **Usage:** Uses `Ctrl` and `Alt` (Meta) key combinations (e.g., `Ctrl+x Ctrl+s` to save).

## 5. Shell Variables and Meta Characters

**Variables**

- **User-Defined:** `name="John"` (No spaces around `=`). Access with `$name`.
- **Environment (Global):** Variables that affect the whole system.

- `$HOME` (User's home directory).
- `$PATH` (Where shell looks for commands).
- `$USER` (Current username).

- **Positional Parameters (Arguments):**
  - `$0`: Name of the script.
  - `$1`: First argument passed to the script.
  - `$#`: Total number of arguments.
  - `$?`: Exit status of the last command (0 = Success).

**Shell Meta Characters (Wildcards)**

Special characters interpreted by the shell before execution.

- `*`: Matches any string. (`rm *.txt` removes all text files).
- `?`: Matches any single character.
- `[ ]`: Matches a range of characters. (`[a-z]` matches any lowercase letter).
- `\`: Escape character. Removes the special meaning of the next char.
- `' '` vs `" "`:
  - **Single Quotes:** strict quoting. `$VAR` is treated as text.
  - **Double Quotes:** weak quoting. `$VAR` is expanded to its value.

## 6. Shell Control Statements (Logic & Loops)

**Shell control statements** are used to control the flow of execution in a shell script. They include **decision-making (logic)** and **looping constructs**.

### A. Decision-Making (Logic Statements)

### 1. if Statement

Used to execute commands when a condition is true.

**Syntax**

```
if condition
then
   commands
fi
```

**Example**

```
if [ $age -ge 18 ]
then
   echo "Eligible to vote"
fi
```

### 2. if-else Statement

```
if [ $num -gt 0 ]
then
   echo "Positive number"
else
```

```
   echo "Negative or Zero"
fi
```

### 3. if-elif-else Statement

```
if [ $marks -ge 90 ]
then
   echo "Grade A"
elif [ $marks -ge 75 ]
then
   echo "Grade B"
else
   echo "Grade C"
fi
```

### 4. case Statement

Used for multiple conditions (alternative to if-elif).

**Syntax**

```
case variable in
   pattern1) commands ;;
   pattern2) commands ;;
   *) default commands ;;
esac
```

**Example**

```
case $day in
   Mon) echo "Start of week" ;;
   Fri) echo "Weekend soon" ;;
   *) echo "Regular day" ;;
esac
```

## B. Looping Statements

### 1. for Loop

Used when the number of iterations is known.

**Syntax**

```
for var in list
do
   commands
done
```

**Example**

```
for i in 1 2 3 4 5
do
   echo "Number: $i"
done
```

### 2. while Loop

Executes as long as the condition is true.

**Syntax**

```
while condition
do
   commands
done
```
**Example**

```
count=1
while [ $count -le 5 ]
do
   echo $count
   count=$((count+1))
done
```

### 3. until Loop

Executes until the condition becomes true.

**Syntax**

```
until condition
do
   commands
done
```
**Example**

```
num=1
until [ $num -gt 5 ]
do
   echo $num
   num=$((num+1))
done
```

### 4. select Loop

Used to create menus.

**Example**

```
select option in Start Stop Restart Exit
do
   case $option in
      Start) echo "Starting..." ;;
      Stop) echo "Stopping..." ;;
      Restart) echo "Restarting..." ;;
      Exit) break ;;
   esac
done
```

## C. Loop Control Statements

**break**

Exits the loop immediately.

```
for i in {1..10}
do
   if [ $i -eq 5 ]
   then
```

```
    break
  fi
  echo $i
done
```

**continue**

Skips current iteration and moves to next.

```
for i in {1..5}
do
  if [ $i -eq 3 ]
  then
    continue
  fi
  echo $i
done
```

## 7. Shell Scripts

A **shell script** is a text file containing a sequence of shell commands that are executed automatically.

**Features**

- Automates repetitive tasks
- Can include variables, loops, conditions, and functions
- Executed by a shell (e.g., bash, sh)

**Example**

```
#!/bin/bash
echo "Hello, World!"
date
```

**How to Run**

```
chmod +x script.sh
./script.sh
```

or

```
bash script.sh
```

## 8. Shell Keywords

**Shell keywords** are **reserved words** that have special meaning to the shell.
They are part of the shell's syntax and **cannot be used as command names**.

**Common Shell Keywords**

| Keyword | Purpose |
|---|---|
| if, then, else, fi | Conditional execution |
| for, while, until, do, done | Loops |
| case, esac | Multi-way branching |

| | |
|---|---|
| function | Define a function |
| select | Menu selection |
| in | Used in loops |
| time | Measure command execution time |

**Example**

```
if [ -f file.txt ]; then
    echo "File exists"
fi
```

## 9. Shell Built-in Commands

**Built-in commands** are commands **implemented inside the shell itself**, not external programs.

**Why Built-ins Exist**

- Faster execution
- Can modify the shell environment (external commands cannot)

**Common Built-in Commands**

| Command | Description |
|---|---|
| cd | Change directory |
| echo | Display output |
| read | Read input |
| exit | Exit shell |
| export | Set environment variables |
| set, unset | Manage variables |
| alias, unalias | Command shortcuts |
| history | Command history |
| jobs, fg, bg | Job control |

**Example**

```
cd /home/user
echo "Welcome"
read name
echo "Hello $name"
```

## 10. Tips and Traps

**Traps (Common Mistakes to Avoid):**

1. **The Space Trap:**

- *Wrong:* `count = 5` (Shell thinks `count` is a command and `=` is an argument).
- *Right:* `count=5` (No spaces in assignment).
- *Right:* `if [ $a -eq 5 ]` (Spaces are **required** inside the brackets `[ ]`).

2. **The Execute Trap:**
   - Writing a script isn't enough. You cannot run it until you grant permission.
   - *Fix:* `chmod +x myscript.sh`
3. **The Shebang Trap:**
   - If you don't write `#!/bin/bash` at the very top (Line 1), the system might use the wrong shell (like plain `sh`) and your advanced code won't work.

## 11. Handling Documents

A "Here Document" (Heredoc) is a way to redirect a block of code or text into a command/ script without using an external file. It tells the shell: *"Read input from here, right now, until you see a specific stopping word."*

**Syntaxz**

```
command << DELIMITER
Text Line 1
Text Line 2
DELIMITER
```

Real-World Example:

Imagine you want to create a welcome note automatically inside a script.

# Instead of creating a file separately, we do it in the script:

```
cat > welcome.txt << ENDOFTEXT
Hello User,
Welcome to the Linux System.
Please do not share your password.
ENDOFTEXT
```

- The shell reads everything after `<< ENDOFTEXT` as input for `cat`.
- It stops when it sees the word `ENDOFTEXT` again.

## 12. C Language Programming in Linux

Linux is the native environment for C programming. The operating system itself is written in C. Developing in Linux follows a strict **Life Cycle**:

### Phase 1: Prototyping

**Definition:** Prototyping is the act of declaring a function's "signature" before defining its actual body/code.

- **Theory:** In C, the compiler reads code from top to bottom. If you use a function (like `calculate_sum`) inside `main()` *before* you have written the code for it, the compiler will error out. A prototype tells the compiler: *"Hey, trust me, this function exists, and here are the inputs it takes."*

**Example:**

```
#include <stdio.h>
```

```c
// --- PROTOTYPE ---
// Tells compiler: "expect a function named add that takes 2 ints"
int add(int a, int b);

int main() {
    int result = add(5, 10); // Compiler is happy now
    printf("%d", result);
    return 0;
}

// --- DEFINITION ---
// The actual code
int add(int a, int b) {
    return a + b;
}
```

**Phase 2: Coding**

Writing the source code in a text editor.

- **Tools:** In Linux, we use editors like vi, vim, nano, or emacs.
- **File Extension:** C files must end with .c (e.g., program.c). Header files end with .h.
- **Pro Tip:** In Linux, header files (like stdio.h) are actually stored in the /usr/include directory. You can actually go there and read them!

**Phase 3: Compiling**

Compiling is the process of converting human-readable Source Code (High-Level Language) into Machine Code (Binary) that the CPU understands.

- **The Tool: GCC** (GNU Compiler Collection).

The 4 Hidden Stages of Compilation:

When you run the command gcc filename.c, the system actually runs four separate programs in a row. This is a favorite exam question.

1. **Preprocessing (The Preprocessor):**
    - **Input:** Source code (.c)
    - **Action:** It handles lines starting with #.
        - Removes comments (// or /* */).
        - Expands Macros (#define PI 3.14 becomes 3.14).
        - Includes Headers (#include <stdio.h> copies the actual file content into your code).
    - **Output:** Expanded source code (.i).
2. **Compilation (The Compiler):**
    - **Input:** Expanded source (.i)
    - **Action:** It checks for **Syntax Errors** (missing semicolons, wrong types). If the code is correct, it translates C code into **Assembly Language**.
    - **Output:** Assembly code (.s).
3. **Assembly (The Assembler):**
    - **Input:** Assembly code (.s)

- **Action:** Translates Assembly mnemonics (like `MOV`, `ADD`, `JMP`) into pure Machine Code (0s and 1s).
- **Output:** Object file (`.o`). The code is binary now, but it cannot run yet because it doesn't know where `printf` is.

4. **Linking (The Linker):**
   - **Input:** Object file (`.o`) + System Libraries.
   - **Action:** It combines your code with the Operating System libraries (like `glibc`). It resolves "unresolved symbols" (like finding the actual code for `printf`).
   - **Output:** Executable file (`a.out` or custom name).

**Commands:**

- `gcc program.c` (Creates `a.out`).
- `gcc -o myapp program.c` (Creates an executable named `myapp`).
- `gcc -Wall program.c` (Shows **All Warnings** – Pro practice).

**Phase 4: Testing**

Running the program with various inputs to ensure it behaves as expected.

- **Execution Command:** `./myapp` (The `./` tells Linux to look in the *current directory*).
- **Types of Testing:**
  - **Happy Path:** Entering valid data (e.g., entering "5" for age).
  - **Edge Cases:** Entering "0" or "-5" for age.
  - **Stress Testing:** Entering huge numbers.

**Phase 5: Debugging**

The process of identifying and removing errors (bugs).

**Types of Errors:**

1. **Compile-Time Errors:** Syntax mistakes (missing `;`). The compiler catches these.
2. **Run-Time Errors:** The program crashes while running (e.g., **Segmentation Fault** / Core Dump). This usually happens when you try to access memory that doesn't belong to you (bad pointers).
3. **Logical Errors:** The program runs but gives the wrong answer (e.g., `2 + 2 = 5`).

The Tool: GDB (GNU Debugger)

GDB is a command-line tool that lets you pause your program while it is running and look inside the memory.

**How to Debug like a Pro:**

1. **Compile with Debug Info:** You must add the `-g` flag.
   - `gcc -g program.c`
2. **Start GDB:**
   - `gdb a.out`
3. **Set a Breakpoint:** Tell GDB to stop at the `main` function.

    - (gdb) break main
4. **Run the program:**
    - (gdb) run
5. **Step-by-Step:**
    - (gdb) next (Executes the next line).
    - (gdb) step (If the line is a function, it goes *inside* the function).
6. **Inspect Variables:**
    - (gdb) print x (Shows the current value of variable x).
7. **Find the Crash:**
    - If the program crashes, type (gdb) bt (Backtrace). It will show you exactly which line caused the crash.

## 13. Filters in Linux

A Filter is a command that accepts text data from Standard Input, transforms/processes it, and sends the result to Standard Output. They are designed to be used with **Pipes (|)**.

### A. grep (Global Regular Expression Print)

The most important filter. It searches for specific words or patterns.

- **Syntax:** grep [options] "pattern" filename
- **Pro Options:**
    - -i: Ignore case (treats "Apple" and "apple" as the same).
    - -v: Invert match (show lines that do **NOT** contain the word).
    - -n: Show line numbers.
    - -c: Count the number of matches.
    - -r: Recursive search (search in all files inside a folder).
- **Example:** grep -i "error" server.log

### B. sort

Arranges lines of text in order.

- **Syntax:** sort [options] filename
- **Pro Options:**
    - (Default): Sorts alphabetically (ASCII order).
    - -n: Numeric sort (Correctly sorts 1, 2, 10 instead of 1, 10, 2).
    - -r: Reverse order (Z to A).
    - -k: Sort by a specific column.
- **Example:** ls -l | sort -nk 5 (Sort files by size, smallest to largest).

### C. uniq (Unique)

Removes **consecutive** duplicate lines.

- **Trap:** `uniq` only detects duplicates if they are next to each other. You must always `sort` first!
- **Syntax:** `sort names.txt | uniq`
- **Option:** `uniq -c` (Counts how many times each line appeared).

**D. `cut`**

Used to slice a file vertically (extract columns).

- **Syntax:** `cut -d "delimiter" -f field_number filename`
- **Example:** Imagine a CSV file `data.txt`: `John,25,Engineer`
  - `cut -d "," -f 1 data.txt` -> Output: `John`
  - `cut -d "," -f 1,3 data.txt` -> Output: `John,Engineer`

**E. `tr` (Translate)**

Used to replace or delete characters. It only works on Streams (stdin), not files directly.

- **Syntax:** `command | tr "old" "new"`
- **Example:** `echo "hello" | tr "a-z" "A-Z"` -> Output: `HELLO`
- **Delete:** `echo "Hello 123" | tr -d "0-9"` -> Output: `Hello` (Deletes numbers).

**F. `wc` (Word Count)**

Counts the contents of a file.

- `-l`: Count lines.
- `-w`: Count words.
- `-c`: Count characters/bytes.

# UNIT - 5

**Linux System Administrations:** File listings, Ownership and Access Permissions, File and Directory types, Managing Files, User and its Home Directory, Booting and Shutting down (Boot Loaders, LILO, GRUB, Bootstrapping, init Process, System services).

## 1. File Listings

In Linux, everything is treated as a file. The ls (list) command is used to view files and directories.

### Common ls Options

- ls: Simple list of filenames.
- ls -a: Shows **all** files, including hidden files (those starting with a dot ., like .bashrc).
- ls -l: Displays a **long listing** format, providing detailed attributes.
- ls -R: Recursive listing (shows subdirectories and their contents).

### Understanding ls -l Output

When you run ls -l, the output looks like this:

drwxr-xr-x 2 student class 4096 Nov 5 10:00 Documents

**Breakdown of columns:**

1. **File Type & Permissions (drwxr-xr-x):** The first character indicates the type (d for directory, - for file). The next 9 characters are permissions.
2. **Link Count (2):** Number of hard links to this file.
3. **Owner (student):** The username of the file owner.
4. **Group (class):** The group name associated with the file.
5. **Size (4096):** File size in bytes.
6. **Modification Date/Time (Nov 5 10:00):** Last time the file was modified.
7. **Filename (Documents):** The name of the file or directory.

## 2. Ownership and Access Permissions

Linux is a multi-user system, so access control is critical. Every file has three categories of users:

1. **User (u):** The owner of the file.
2. **Group (g):** Other users in the file's group.
3. **Others (o):** Everyone else on the system.

### Permission Types

| Permission | Permission | Permission | Permission |
|---|---|---|---|
| **Read** | r | Can view content (cat) | Can list contents (ls) |

| | | | |
|---|---|---|---|
| **Write** | w | Can modify/ delete content | Can create/delete files inside it |
| **Execute** | x | Can run file as a program | Can enter directory (cd) |

**Changing Permissions (chmod)**

You can change permissions using **Symbolic Mode** or **Absolute (Octal) Mode**.

**A. Symbolic Mode**

Syntax: chmod [who][operator][permission] filename

- **Who:** u (user), g (group), o (others), a (all).
- **Operator:** + (add), - (remove), = (set).
- **Example:**
  - chmod u+x script.sh (Add execute permission for the owner).
  - chmod go-w file.txt (Remove write permission for group and others).

**B. Numerical (Octal) Mode [Important for Exams]**

Permissions are represented by octal numbers.

- **Read (r) = 4**
- **Write (w) = 2**
- **Execute (x) = 1**
- **No Permission (-) = 0**

Calculation Formula:

Sum the values for each user category.

**Example 1: Full permissions for owner, read-execute for group/others.**

- **Owner:** r + w + x = 4 + 2 + 1 = **7**
- **Group:** r + - + x = 4 + 0 + 1 = **5**
- **Others:** r + - + x = 4 + 0 + 1 = **5**
- **Command:** chmod 755 filename

**Example 2: Read-write for owner, Read-only for group, No access for others.**

- **Owner:** r + w + - = 4 + 2 + 0 = **6**
- **Group:** r + - + - = 4 + 0 + 0 = **4**
- **Others:** - + - + - = 0 + 0 + 0 = **0**
- **Command:** chmod 640 filename

**Changing Ownership**

- **chown (Change Owner):** Changes the file owner.
  - *Example:* chown user1 file.txt
- **chgrp (Change Group):** Changes the file group.

- *Example:* `chgrp staff file.txt`

## 3. File and Directory Types

Linux identifies file types by the first character in the permission string (from `ls -l`).

1. **Regular File (`-`):** Standard files (text, images, binaries).
2. **Directory (`d`):** Contains references to other files.
3. **Link (`l`):** A pointer to another file (like a shortcut).
4. **Character Special File (`c`):** Hardware devices that handle data byte-by-byte (e.g., terminals, keyboards).
5. **Block Special File (`b`):** Hardware devices that handle data in blocks (e.g., Hard drives, USBs).
6. **Socket (`s`):** Used for inter-process communication (network sockets).
7. **Named Pipe (`p`):** Also called FIFO, used to pass data between running processes.

## 4. Managing Files and User Directories

**File Management Commands**

- **Create File:** `touch filename` (creates empty file) or `cat > filename`.
- **Create Directory:** `mkdir dirname`.
- **Copy:** `cp source destination` (Use `cp -r` for directories).
- **Move/Rename:** `mv oldname newname` or `mv file path/`.
- **Remove:**
  - `rm filename` (delete file).
  - `rmdir dirname` (delete *empty* directory).
  - `rm -rf dirname` (force delete directory and contents—use with caution!).

**User and Home Directory**

- **Root Directory (`/`):** The top of the file system hierarchy.
- **Home Directory (`/home/username`):** Where a standard user stores personal data.
  - The symbol `~` (tilde) represents the current user's home directory.
- **Root User's Home:** Located at `/root` (distinct from `/`).

## 5. Booting and Shutting Down

The Linux boot process (Bootstrapping) involves a sequence of 6 strictly defined stages.

**Stage 1: BIOS (Basic Input/Output System)**

- Performs **POST** (Power-On Self-Test) to check hardware (RAM, CPU, Keyboard).
- Searches for the bootable device (Hard Disk, CD, USB).
- Loads the **MBR** (Master Boot Record).

**Stage 2: MBR / Boot Loader**

The MBR is the first 512 bytes of the hard drive. It contains the boot loader.

- **Boot Loader's Job:** Load the OS kernel into memory.
- **Types of Boot Loaders:**
    1. **LILO (Linux Loader):**
        - *Legacy/Old.*
        - It depended on the physical location of the kernel on the disk.
        - If you updated the kernel, you had to manually run the `lilo` command to update the map.
        - No interactive command line.
    2. **GRUB (GRand Unified Bootloader):**
        - *Modern/Standard.*
        - Understands file systems, so it can find the kernel by filename.
        - Highly configurable (menu to select OS, recovery mode).
        - **Stage 1:** Loads tiny code from MBR.
        - **Stage 2:** Loads the GRUB configuration menu (`/boot/grub/grub.cfg`).

**Stage 3: Kernel**

- The Boot loader loads the **Kernel** (core of the OS).
- The Kernel initializes hardware drivers and mounts the **Root File System** as "read-only" initially.

**Stage 4: Init Process**

- The Kernel starts the first program: **`/sbin/init`**.
- This process always has **PID = 1** (Process ID).
- It is the parent of all other processes.

**Stage 5: Runlevels / System Services**

The `init` process looks at the configuration to decide what state (Runlevel) the system should be in.

- **SysVinit (Older):** Uses `/etc/inittab`.
    - *Runlevel 0:* Halt/Shutdown.
    - *Runlevel 1:* Single-user mode (Maintenance).
    - *Runlevel 3:* Multi-user mode (Text only/Server).
    - *Runlevel 5:* Graphical mode (GUI).
    - *Runlevel 6:* Reboot.
- **Systemd (Newer):** Uses "Targets" instead of runlevels (e.g., `graphical.target`).

**Stage 6: User Login**

- `getty` or GUI Display Manager starts.
- User enters credentials.
- Shell (e.g., Bash) is launched.

**Shutting Down**

Commands to safely stop the system:

- `shutdown -h now`: Halts the system immediately.
- `shutdown -r now`: Reboots immediately.
- `init 0`: Switches to Runlevel 0 (Shutdown).
- `halt`: Stops the CPU.

**Numerical Example for Exam**

**Question:** Calculate the Octal Permission code for a file where:

- **Owner** can Read, Write, and Execute.
- **Group** can Read and Execute.
- **Others** can only Read.

**Answer:**

1. **Owner (rwx):**
   - Read (4) + Write (2) + Execute (1) = **7**
2. **Group (r-x):**
   - Read (4) + Write (0) + Execute (1) = **5**
3. **Others (r--):**
   - Read (4) + Write (0) + Execute (0) = **4**

Result: The permission code is 754.

Command: chmod 754 filename